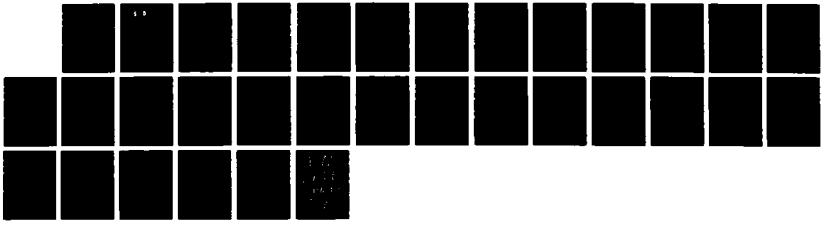
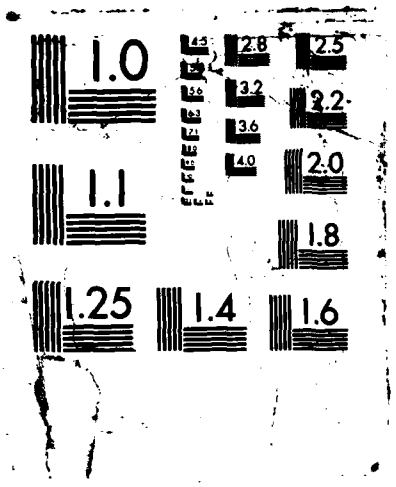
AD-A190 087 SUPERCOMPUTER PROGRAMMING ENVIRONMENTS(U) ILLINOIS UNIV 1/1
AT URBANA CENTER FOR SUPERCOMPUTING RESEARCH AND
DEVELOPMENT D A PADUA ET AL. 30 OCT 87 CSRD-673
UNCLASSIFIED AFOSR-TR-87-1987 F49620-86-C-0136 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(2)

SEC

AD-A190 887 ORT DOCUMENTATION PAGE

DTIC FILE COPY

DTIC ELECTE JAN 25 1988 S D

1a.	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTH[illegible]	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRA[illegible] SCHEDULE	
4. PERFORMING ORGANIZATION R[illegible] NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR.TR. 87-1087
6a. NAME OF PERFORMING ORGANIZATION: The Board of Trustees of the University of Illinois	6b. OFFICE SYMBOL (If applicable):
6c. ADDRESS (City, State, and ZIP Code): 506 S. Wright St. Urbana, IL 61801	7a. NAME OF MONITORING ORGANIZATION: AFOSR/NM
	7b. ADDRESS (City, State, and ZIP Code): AFOSR/NM Bldg 410 Bolling AFB DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR	8b. OFFICE SYMBOL (If applicable): NM
	9. PROCUREMENT INSTRUMENT[illegible] IFICATION NUMBER: F49620-86-C-0136
8c. ADDRESS (City, State, and ZIP Code): AFOSR/NM Bldg 410 Bolling AFB DC 20332-6448	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO. A3; WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Supercomputer Programming Environments

12. PERSONAL AUTHOR(S)
David A. Padua, Vincent A. Guarna Jr., Duncan H. Lawrie

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Publication	FROM 10/1/86 TO 9/30/87	Oct 30, 87	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The quest to apply an ever-increasing amount of computing power to numerical applications has resulted in the evolution of a broad spectrum of ideas and implementations for high performance computing systems. The architectural complexity of these high performance systems typically requires special tools and techniques to achieve efficeient utilization of available computational resources. These tools range from automatic restructuring and optimizing compilers to interactive debugging and performance analysis systems. The programming environment for these systems must be general and adaptive, providing the appropriate level of assistance for users of varying levels of sophistication. This paper presents recent developments in supercomputer environments, and focuses in more detail on the Cedar Project which is currently under way at the University of Illinois Center for Supercomputing Research and Developement. The Cedar Project consists of the construction of a prototype multiprocessor, restructuring compilers for the Fortran and C programming languages, and an integrated graphics-based programming environment intended to serve the needs of scientific applications users.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS		
22a. NAME OF RESPONSIBLE INDIVIDUAL: Maj. John Thomas	22b. TELEPHONE (Include Area Code): (202) 767-5026	22c. OFFICE SYMBOL: NM

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

INSTRUCTIONS FOR PREPARATION OF REPORT DOCUMENTATION PAGE

GENERAL INFORMATION

The accuracy and completeness of all information provided in the DD Form 1473, especially classification and distribution limitation markings, are the responsibility of the authoring or monitoring DoD activity.

Because the data input on this form will be what others will retrieve from DTIC's bibliographic data base or may determine how the document can be accessed by future users, care should be taken to have the form completed by knowledgeable personnel. For better communication and to facilitate more complete and accurate input from the originators of the form to those processing the data, space has been provided in Block 22 for the name, telephone number, and office symbol of the DoD person responsible for the input cited on the form.

All information on the DD Form 1473 should be typed.

Only information appearing on or in the report, or applying specifically to the report in hand, should be reported. If there is any doubt, the block should be left blank.

Some of the information on the forms (e.g., title, abstract) will be machine indexed. The terminology used should describe the content of the report or identify it as precisely as possible for future identification and retrieval.

NOTE: **Unclassified abstracts and titles describing classified documents may appear separately from the documents in an unclassified context, e.g., in DTIC announcement bulletins and bibliographies. This must be considered in the preparation and marking of unclassified abstracts and titles.**

The Defense Technical Information Center (DTIC) is ready to offer assistance to anyone who needs and requests it. Call Data Base Input Division, Autovon 284-7044 or Commercial (202) 274-7044.

SECURITY CLASSIFICATION OF THE FORM

In accordance with DoD 5200.1-R, Information Security Program Regulation, Chapter IV Section 2, paragraph 4-200, classification markings are to be stamped, printed, or written at the top and bottom of the form in capital letters that are larger than those used in the text of the document. See also DoD 5220.22-M, Industrial Security Manual for Safeguarding Classified Information, Section II, paragraph 11a(2). This form should be unclassified, if possible.

SPECIFIC BLOCKS

Block 1a. Report Security Classification: Designate the highest security classification of the report. (See DoD 5220.1-R, Chapters I, IV, VII, XI, Appendix A.)

Block 1b. Restricted Marking: Enter the restricted marking or warning notice of the report (e.g., CNWDI, RD, NATO).

Block 2a. Security Classification Authority: Enter the commonly used markings in accordance with DoD 5200.1-R, Chapter IV, Section 4, paragraph 4-400 and 4-402. Indicate classification authority.

Block 2b. Declassification / Downgrading Schedule: Indicate specific date or event for declassification or the notation, "Originating Agency Determination Required" or "OADR." Also insert (when applicable) downgrade to _____ on _____ (e.g., Downgrade to Confidential on 6 July 1983). (See also DoD 5220.22-M, Industrial Security Manual for Safeguarding Classified Information, Appendix II.)

NOTE: **Entry must be made in Blocks 2a and 2b except when the original report is unclassified and has never been upgraded.**

Block 3. Distribution/Availability Statement of Report: Insert the statement as it appears on the report. If a limited distribution statement is used, the reason must be one of those given by DoD Directive 5200.20, Distribution Statements on Technical Documents, as supplemented by the 18 OCT 1983 SECDEF Memo, "Control of Unclassified Technology with Military Application." The Distribution Statement should provide for the broadest distribution possible within limits of security and controlling office limitations.

Block 4. Performing Organization Report Number(s): Enter the unique alphanumeric report number(s) assigned by the organization originating or generating the report from its research and whose name appears in Block 6. These numbers should be in accordance with ANSI STD 239.23-74, "American National Standard Technical Report Number." If the Performing Organization is also the Monitoring Agency, enter the report number in Block 4.

Block 5. Monitoring Organization Report Number(s): Enter the unique alphanumeric report number(s) assigned by the Monitoring Agency. This should be a number assigned by a DoD or other government agency and should be in accordance with ANSI STD 239.23-74. If the Monitoring Agency is the same as the Performing Organization, enter the report number in Block 4 and leave Block 5 blank.

Block 6a. Name of Performing Organization: For in-house reports, enter the name of the performing activity. For reports prepared under contract or grant, enter the contractor or the grantee who generated the report and identify the appropriate corporate division, school, laboratory, etc., of the author.

Block 6b. Office Symbol: Enter the office symbol of the Performing Oganization.

Block 6c. Address: Enter the address of the Performing Organization. List city, state, and ZIP code.

Block 7a. Name of Monitoring Organization: This is the agency responsible for administering or monitoring a project, contract, or grant. If the monitor is also the Performing Organization, leave Block 7a. blank. In the case of joint sponsorship, the Monitoring Organization is determined by advance agreement. It can be either an office, a group, or a committee representing more than one activity, service, or agency.

Block 7b. Address: Enter the address of the Monitoring Organization. Include city, state, and ZIP code.

Block 8a. Name of Funding/Sponsoring Organization: Enter the full official name of the organization under whose immediate funding the document was generated, whether the work was done in-house or by contract. If the Monitoring Organization is the same as the Funding Organization, leave 8a blank.

Block 8b. Office Symbol: Enter the office symbol of the Funding/Sponsoring Organization.

Block 8c. Address: Enter the address of the Funding/ Sponsoring Organization. Include city, state and ZIP code.

Block 9. Procurement Instrument Identification Number: For a contractor grantee report, enter the complete contract or grant number(s) under which the work was accomplished. Leave this block blank for in-house reports.

Block 10. Source of Funding (Program Element, Project, Task Area, and Work Unit Number(s): These four data elements relate to the DoD budget structure and provide program and/or administrative identification of the source of support for the work being carried on. Enter the program element, project, task area, work unit accession number, or their equivalents which identify the principal source of funding for the work required. These codes may be obtained from the applicable DoD forms such as the DD Form 1498 (Research and Technology Work Unit Summary) or from the fund citation of the funding instrument. If this information is not available to the authoring activity, these blocks should be filled in by the responsible DoD Official designated in Block 22. If the report is funded from multiple sources, identify only the Program Element and the Project, Task Area, and Work Unit Numbers of the principal contributor.

Block 11. Title: Enter the title in Block 11 in initial capital letters exactly as it appears on the report. Titles on all classified reports, whether classified or unclassified, must be immediately followed by the security classification of the title enclosed in parentheses. A report with a classified title should be provided with an unclassified version if it is possible to do so without changing the meaning or obscuring the contents of the report. Use specific, meaningful words that describe the content of the report so that when the title is machine-indexed, the words will contribute useful retrieval terms.

If the report is in a foreign language and the title is given in both English and a foreign language, list the foreign language title first, followed by the English title enclosed in parentheses. If part of the text is in English, list the English title first followed by the foreign language title enclosed in parentheses. If the title is given in more than one foreign language, use a title that reflects the language of the text. If both the text and titles are in a foreign language, the title should be translated, if possible, unless the title is also the name of a foreign periodical. Transliterations of often used foreign alphabets (see Appendix A of MIL-STD-8478) are available from DTIC in document AD-A080 800.

Block 12. Personal Author(s): Give the complete name(s) of the author(s) in this order: last name, first name, and middle name. In addition, list the affiliation of the authors if it differs from that of the performing organization.

List all authors. If the document is a compilation of papers, it may be more useful to list the authors with the titles of their papers as a contents note in the abstract in Block 19. If appropriate, the names of editors and compilers may be entered in this block.

Block 13a. Type of Report: Indicate whether the report is summary, final, annual, progress, interim, etc.

Block 13b. Time Covered: Enter the inclusive dates (*year, month, day*) of the period covered, such as the life of a contract in a final contractor report.

Block 14. Date of Report: Enter the year, month, and day, or the year and the month the report was issued as shown on the cover.

Block 15. Page Count: Enter the total number of pages in the report that contain information, including cover, preface, table of contents, distribution lists, partial pages, etc. A chart in the body of the report is counted even if it is unnumbered.

Block 16. Supplementary Notation: Enter useful information about the report in hand, such as: "Prepared in cooperation with...," "Translation at (or by)...," "Symposium..." If there are report numbers for the report which are not noted elsewhere on the form (such as internal series numbers or participating organization report numbers) enter in this block.

Block 17. COSATI Codes: This block provides the subject coverage of the report for announcement and distribution purposes. The categories are to be taken from the "COSATI Subject Category List" (DoD Modified), Oct 65, AD-624 000. A copy is available on request to any organization generating reports for DoD. At least one entry is required as follows:

Field - to indicate subject coverage of report.

Group - to indicate greater subject specificity of information in the report.

Sub-Group - if specificity greater than that shown by Group is required, use further designation as the numbers after the period (.) in the Group breakdown. Use only the designation provided by AD-624 000.

Example: The subject "Solid Rocket Motors" is Field 21, Group 08, Subgroup 2 (page 32, AD-624 000).

Block 18. Subject Terms: These may be descriptors, keywords, posting terms, identifiers, open-ended terms, subject headings, acronyms, code words, or any words or phrases that identify the principal subjects covered in the report, and that conform to standard terminology and are exact enough to be used as subject index entries. Certain acronyms or "buzz words" may be used if they are recognized by specialists in the field and have a potential for becoming accepted terms. "Laser" and "Reverse Osmosis" were once such terms.

If possible, this set of terms should be selected so that the terms individually and as a group will remain UNCLASSIFIED without losing meaning. However, priority must be given to specifying proper subject terms rather than making the set of terms appear "UNCLASSIFIED." Each term on classified reports must be immediately followed by its security classification, enclosed in parentheses.

For reference on standard terminology the "DTIC Retrieval and Indexing Terminology" DRIT-1979, AD-A068 500, and the DoD "Thesaurus of Engineering and Scientific Terms (TEST) 1968, AD-672 000, may be useful.

Block 19. Abstract: The abstract should be a pithy, brief (preferably not to exceed 300 words), factual summary of the most significant information contained in the report. However, since the abstract may be machine-searched, all specific and meaningful words and phrases which express the subject content of the report should be included, even if the word limit is exceeded.

If possible, the abstract of a classified report should be unclassified and consist of publicly releasable information (Unlimited), but in no instance should the report content description be sacrificed for the security classification.

NOTE: **An unclassified abstract describing a classified document may appear separately from the document in an unclassified context e.g., in DTIC announcement or bibliographic products. This must be considered in the preparation and marking of unclassified abstracts.**

For further information on preparing abstracts, employing scientific symbols, verbalizing, etc., see paragraphs 2.1(n) and 2.3(b) in MIL-STD-847B.

Block 20. Distribution / Availability of Abstract: This block must be completed for all reports. Check the applicable statement: "unclassified / unlimited," "same as report," or, if the report is available to DTIC registered users " DTIC users."

Block 21. Abstract Security Classification: To ensure proper safeguarding of information, this block must be completed for all reports to designate the classification level of the entire abstract. For CLASSIFIED abstracts, each paragraph must be preceded by its security classification code in parentheses.

Block 22a,b,c. Name, Telephone and Office Symbol of Responsible Individual: Give name, telephone number, and office symbol of DoD person responsible for the accuracy of the completion of this form.

☆U.S. Government Printing Office: 1984—430-870/10040

CSRD report no. 673

AFOSR-TR- 87-1987

Center for
Supercomputing Research and Development

Supercomputer Programming Environments

David A. Padua
Vincent A. Guarna, Jr.
Duncan H. Lawrie

June 9, 1987

University of Illinois at Urbana–Champaign
104 S. Wright Street
Urbana, Illinois 61801
(217) 333-6223 %uicsrd@a.cs.uiuc.edu

Version: 2384-June 9, 1987
Produced: Tue Jun 9 13:15:34 CDT 1987

OUTLINE

1 ABSTRACT 1
2 INTRODUCTION 1
3 ISSUES IN PARALLEL PROGRAMMING LANGUAGES 3
4 STATE-OF-THE-ART PROGRAMMING TOOLS 8
Restructuring and Interactive Restructurers 8
Debuggers 13
Performance Evaluation 16
Integration 19
5 ACKNOWLEDGEMENT 22
6 REFERENCES 22

CSRD report no. 673

Supercomputer Programming Environments

David A. Padua
Vincent A. Guarna, Jr.
Duncan H. Lawrie

June 9, 1987

Abstract

The quest to apply an ever-increasing amount of computing power to numerical applications has resulted in the evolution of a broad spectrum of ideas and implementations for high performance computing systems. The architectural complexity of these high performance systems typically requires special tools and techniques to achieve efficient utilization of available computational resources. These tools range from automatic restructuring and optimizing compilers to interactive debugging and performance analysis systems. The programming environment for these systems must be general and adaptive, providing the appropriate level of assistance for users of varying levels of sophistication. This paper presents recent developments in supercomputer environments, and focuses in more detail on the Cedar Project which is currently under way at the University of Illinois Center for Supercomputing Research and Development. The Cedar Project consists of the construction of a prototype multiprocessor, restructuring compilers for the Fortran and C programming languages, and an integrated graphics-based programming environment intended to serve the needs of scientific applications users.

Keywords: *scientific computation, parallel computation, parallel languages, vector languages, programming environments, optimizing compilers, parallel debuggers*

This paper will appear in the Proceedings of the Symposium on Parallel Computations and Their Impact on Mechanics, to be held at the ASME Winter Annual meeting in Boston, December 13-18, 1987.

This work was supported in part by the National Science Foundation under Grant Nos. US NSF DCR84-06916 and US NSF DCR84-10110, the US Department of Energy under Grant No. US DOE DE-FG02-85ER25001, the United States Air Force under Grant AFOSR-85-0211 and by a donation from the IBM Corporation.

1 ABSTRACT

The quest to apply an ever-increasing amount of computing power to numerical applications has resulted in the evolution of a broad spectrum of ideas and implementations for high performance computing systems. The architectural complexity of these high performance systems typically requires special tools and techniques to achieve efficient utilization of available computational resources. These tools range from automatic restructuring and optimizing compilers to interactive debugging and performance analysis systems. The programming environment for these systems must be general and adaptive, providing the appropriate level of assistance for users of varying levels of sophistication. This paper presents recent developments in supercomputer environments, and focuses in more detail on the Cedar Project which is currently under way at the University of Illinois Center for Supercomputing Research and Development. The Cedar Project consists of the construction of a prototype multiprocessor, restructuring compilers for the Fortran and C programming languages, and an integrated graphics-based programming environment intended to serve the needs of scientific applications users.

2 INTRODUCTION

Present supercomputers[1] require vectorization of codes to achieve anywhere near their performance potential. Additionally, on some machines, vector registers must be carefully managed to avoid as much memory access as possible lest memory access become a bottleneck. This must all be done while managing disk I/O which is vastly slower than

[1] In this paper we use the word *supercomputer* to refer to the fastest general-purpose scientific computers, e.g., machines like the Cray X-MP, Cray II, CDC Cyber 205, ETA 10, Fujitsu Facom VP, Hitachi S-810, and NEC SX. These machines can have at least several processors all sharing a common memory. Other machines, for example NCUBE, hypercube, etc., have many processors and a distributed (unshared) memory. These latter machines are striving for "super" status, and indeed some of these can provide supercomputer performance on certain applications. However, our expertise lies in the former area and we will not address the latter machines.

the processor and I/O from some form of bulk random access memory.

The next generation of supercomputers will be even more complex than present supercomputers. Machines with multiple processors are already in the field. This will be the primary characteristic of the next generation of supercomputers—multiprocessing[2]—but more so. Yet techniques for using multiprocessors are poorly understood today. Not only do programs need to be rewritten as they were for vector machines to capitalize on performance gains available from multiprocessors, but often the algorithms themselves must be redesigned to allow multiprocessing. Optimizing compilers are just beginning to do a credible job of vectorization. They are a long way from being able to restructure a program to use multiprocessors effectively, a process we call *parallelization*. Further, the design of parallel algorithms is still a relatively new art.

Multiprocessing leads to other difficulties as well. Programming multiple, asynchronous tasks is probably an order of magnitude more difficult than what most of us are used to programming--a single execution stream. Once we start using multiple execution streams, we must be careful about cases where multiple streams access the same data. Where data access by multiple execution streams might cause a problem, we must use some form of synchronization. And in machines lacking a shared memory, data must be explicitly moved from processor to processor (or sometimes, rather than moving the data, in effect the program is moved from processor to processor).

Debugging parallel programs is also an order of magnitude more difficult. For example, most errors are not easily reproduced because the exact time ordering of the multiple execution streams will vary from one run to the next. Thus, errors caused by poor conceptualization of the synchronous/asynchronous nature of the program are not only the easiest errors to make, but the most difficult to find and reproduce.

Memories will also increase in complexity. For example, many new supercomputers will of necessity use cache memories to better match the processor and memory speeds. Consider, however, the effect of vectorization on cache memories. When a program is vectorized, the result is usually statements that compute on whole vectors at a time. Often each vector in such a statement is only accessed once. Yet, if the vector is long (and the longer the better for performance), then it may have the effect of flushing the cache. Already, we see several common optimizations at odds. This same phenomenon adversely affects the locality of programs in machines which have paged memories. But the complexity does not stop there. New machines are likely to contain some mixture of *local memories* (memory accessible by only one processor) and *shared memory* (memory shared by some or all of the processors). To get the full potential performance of these machines, data must be carefully allocated in the best memory depending on its access characteristics. Often this allocation must change during the execution of the program.

Add to all this the increasing disparity between processor speed and I/O speed, and we have even greater need for complexity in how I/O is handled. This in turn necessitates multitracked I/O (streaming data to or from multiple disks simultaneously,) disk caches, and bulk random access memories.

It is perhaps ironic that in this age of the microprocessor and personal computer, when software is finally becoming easy, perhaps even fun to use, supercomputers are becoming more difficult to use. We must see to it that this does not happen. Better programming environments are needed— compilers, languages, debugging and performance tools--if we are to make use of the tremendous potential offered by supercomputers.

---

[2] We define *multiprocessing* to mean the use of more than one processor on one job. This is sometimes called multitasking, and the term *parallel processing* has also come into vogue to mean the same thing. This is different from *vector processing*, which means computing on vectors, usually with a pipeline processor like the Cray series.

3 ISSUES IN PARALLEL PROGRAMMING LANGUAGES

Several approaches are possible in the design and selection of programming languages for parallel processing. In this section we will discuss Fortran and its extensions. A few remarks will be made at the end on alternative languages.

Fortran will be emphasized due to its predominance. It is safe to say that most of the application code for parallel scientific computers is in the form of numerical programs written in Fortran, and that this situation will continue in the near future. Supercomputers use either an optimizing compiler or Fortran extensions to exploit both vector and asynchronous parallelism. We will discuss these two forms of parallelism next, starting with vector parallelism.

Some vendors use standard sequential Fortran and rely on the compiler to exploit vector parallelism. These compilers include a vectorization phase where regular do loops are internally transformed into vector assignment statements. To give the programmer control over what is vectorized and how, these Fortran compilers all accept some form of *vectorization commands* supplied via comment cards. Also, a programming style may be suggested to the programmer to help the vectorizer.[3] The main advantage of using standard sequential Fortran is portability. Thus, Fortran programs (even if they were not written for supercomputers) can often be efficiently run on a new supercomputer either without change or with the addition of a few compiler directives with vectorization commands for the new machine.

Another possible approach to exploit vector parallelism is to extend Fortran with vector assignment statements. Four types of constructs have been used for vector assignment statements. The first construct, *control vectors*, was used by two early vector languages: the Burroughs Illiac IV Fortran (Ref. 11), and Glypnir (Ref. 33). The latter language was also designed for the Illiac IV, and while it was based on Algol, its control structures could be trivially incorporated into Fortran. Control vectors were boolean vectors used to control vector operations. Burroughs Illiac IV Fortran used control vectors as array subscripts. A * denoted a boolean vector with all elements set to true. Thus,

```
Real A(100), B(100)
A(*) = B(*) + A(*)                                        (1)
```

added corresponding elements of arrays A and B and assigned the result to array A. On the other hand,

```
      do 10 i = 1, 100, 2
      M (i) = .true.
      M (i+1) = .false.                                   (2)
10    continue
      A (M(*))=B(M(*)) +A(M(*))
```

did the same thing but only for the odd elements of A and B.

In Glypnir control vectors had 64 elements, one for each Illiac IV processor (PE), which were used to control whether a processor was to execute or remain idle. Variables in Glypnir could be declared to be of the pe type; this specified that there would be a copy of the variable on each processor. To illustrate these ideas consider the following program:

---

[3] For example, the Cray CFT manual suggests: "Keep subscripts simple and explicit; do not use parentheses in subscripts."

```
pe real x,y,z
...                                                    (3)
for all z < 0 do x = y + 1
```

This program specified that x, y and z were 64 element arrays, and that for $1 \leq i \leq 64$, y(i) + 1 was to be assigned to x(i) whenever z(i) < 0. In the for all statement, the ith element of the control vector had the boolean value 'z(i) < 0'.

The language IVTRAN, also developed for the Illiac IV, introduced a second type of construct: do for all (Ref. 38). This construct specified the subscripts to be used in the vector operation. Thus,

```
      do 10 for all i = 1, 100, 2
10    A(i) = B (i) + A (i)                             (4)
```

operated in vector form on the odd elements of A and B as was done by loop (2). The do for all index was not limited to a single dimension. Thus,

```
      real A(100, 50)
      do 10 for all i = [1, 100].c.[1, 50]             (5)
10    A(i) = A(i) + 1
```

added one to each element of the 100 x 50 array A. (The .c. means Cartesian product, and i represents a pair of integers <j,k> where j ∈ [1,100] and k ∈ [1,50].) A construct similar to do for all was present in early versions of the Fortran 8X (Ref. 37) draft standard but has been removed.

The last three vector constructs we will discuss are the ones presently adopted by the Fortran 8X proposal. These were originally developed as part of Vectran (Refs. 42 and 43), an extension to Fortran developed at the IBM Houston Scientific Center. The basic construct is the vector assignment statement based on triplets, three integers separated by colons that specify beginning subscript, ending subscript, and stride.[4] Thus,

```
A(1:100:2) = B(1:100:2) + A(1:100:2)                   (6)
```

performs the operation and assignment on the odd elements of A and B and is equivalent to loops (2) and (4). The triplet notation is complemented with the identify statement used for the selection of array sections like matrix diagonals (which cannot be expressed via triplets), and the where statement used to perform conditional vector element assignments.

The adoption of Fortran 8X will make use of vector constructs more common. However, this will not rule out vectorizing compilers as will be discussed below. Both vector constructs and vectorization will probably coexist as they do today in, for example, Alliant Fortran (Ref. 5).

A second class of constructs are those used to express asynchronous parallelism. In what follows, we will discuss Fortran extensions assuming shared memory (see footnote 1). Extensions to Fortran for systems without shared memory should typically involve just a few intrinsic routines for message passing and synchronization (see, for example, Ref. 2).

Multitasking constructs are the more traditional ones. Generally there is some kind of fork, process, or co-begin statement that causes the start of a new execution stream that can execute in parallel with the original stream. We call this new stream a *process*. Note that the number of processes started may far exceed the number of processors available for parallel execution. However, this only influences performance, since in multitasking the operating system automatically multiplexes processors and therefore gives the illusion of the availability of an unlimited number of processors.

---

[4] *Stride* is the distance between successive array elements.

When multitasking is implemented in software,[5] there can be a substantial amount of overhead involved in setting up a new process because storage must be allocated, etc. Thus, if the granularity of the task[6] is small, i.e., if the amount of work to be done by the process is small, then the overhead of allocating a new process may overwhelm the useful work done. To get around the high overhead of *process* allocation, especially when the tasks are small, *microtasking* is sometimes used. With microtasking, it is not necessary to allocate a complete new process for each task. What usually happens is that the number of processes started is equal to the number of available processors. Then each process will be assigned one task but no multiplexing will be done. In other words, the task will remain associated to the process until it is completed, thus saving some of the overhead associated with process allocation. Remaining tasks are allocated to processes only when a process becomes available by virtue of having finished a task. This assignment of tasks to processes is done by the user (or perhaps the Fortran run-time support library).

Microtasking can cause problems, however. In the case of multitasking, each task has a process. If any process (and thus task) is blocked, then that process is suspended and the operating system automatically switches the processor to any other ready process. For example, suppose we have three tasks, α, β, and γ, and three processes a, b, and c. Further assume that there are only two processors and they start executing processes a (with task α) and b (task β.) If b gets blocked for some reason (for example, waiting for I/O or waiting for a signal from task γ), then b is suspended which releases a processor and allows process c to begin.

Now, suppose we are microtasking. Further suppose that there are two processors again, that the user has asked for two processes, x and y, and that task α is assigned to process x, task β is assigned to y and task γ remains unassigned. Now if β is blocked waiting for a signal from γ and simultaneously α is blocked waiting for a signal from β, then we have a condition known as a deadlock — β and α cannot finish because they need a signal from γ, but γ can never signal because γ cannot start until either α or β have finished. Thus we have a case where, if we allocated a distinct process to each task, no deadlock occurs, whereas if we do microtasking and the number of tasks is greater than the number of processes, we can get a deadlock. Of course, the user (or support library) can design a more clever microtasking system, but this will likely increase the overhead and thus defeat the original reason for microtasking.

Microtasking systems allow only restricted types of synchronization, for example, *critical regions* and *cascade synchronization.* The reason for this restriction is to avoid deadlock situations like the one discussed above. Assume that sections of code in different streams may be identified with a name. The critical region mechanism guarantees that no two processors will be inside a critical section with the same name at the same time. In this case, we say there is *mutual exclusion.* For cascade synchronization it is assumed that if a task τ signals another task μ, then a process will be allocated to τ before it is allocated to μ.

In an attempt to clarify these ideas, we will now discuss loop parallelism. A parallel loop whose iterations contain no synchronization across iterations (except for critical sec-

---

[5] Multitasking is almost always implemented in software. The only exception we know of is the Denelcor HEP (Refs. 48 and 29) where the implementation was in hardware. This made multitasking fast enough to be used to start parallel loops, and obviated the need for microtasking.

[6] In some contexts *task* is considered a synonym of *process.* Here the word task will mean an activity to be performed by a process. This activity may be, for example, to execute a statement, a group of statements, or one iteration of a loop.

tions) is called a doall[7] loop. The defining characteristic of doall loops is that their iterations may be executed in parallel and that processors may be allocated to iterations in any order. An example of such a loop is the following:

```
doall i=1,n
      B(i) = A(i)
      do while (B(i)**2-A(i) .gt. epsilon)
            B(i)=(B(i)+A(i)/B(i))/2.0                    (7)
      end do
end doall
```

Iteration i of this loop computes the square root of A(i) using Newton-Raphson, and assigns it to B(i) (we assume that A(i) > 1).

Doall loops should not be synchronized in such a way that a certain number of processors or a certain processor allocation order would be required for correct execution. For example, the loop:[8]

```
semaphore S(:)
...
V(S(1))
V(T(1))
doall i=1,n

      P(S(i))
      A(i)=A(i-1)+1                                      (8)
      V(S(i+1))

      P(T(i))
      B(i)=B(i-1)+A(i)
      V(T(i+1))

end doall
```

is invalid since iteration i > 1 cannot start execution until iteration i-1 has started, and this imposes an order on processor allocation. Thus, if only two processors were available at run time, and they were allocated to iterations 2 and 3, the program would never complete since iteration 2 cannot start until semaphore S(2) is incremented in iteration 1.

Parallel loops where iterations wait for synchronization signals from previous iterations happen with some frequency. For these types of loops, the doacross construct can be used. This construct requires that processors be allocated first to earlier iterations. Thus, the previous loop with the doall keyword replaced by the doacross will be correct.

Another example of doacross is obtained by transforming the loop:

---

[7] From FMP Fortran (Ref. 13), which used a generalized version of IVTRANs [illegible] vector construct.

[8] In this loop, P and V are the well known synchronization operations. These operate on semaphores. The P(S) operation tests the semaphore S and if its value is greater than zero, it decrements S and proceeds. If S is zero, the process waits until a V(S) operation is executed. The V(S) operation checks whether there are processes waiting on semaphore S; if so, it allows one of them to proceed; otherwise, V(S) increments S by one. A fundamental characteristic of these operations is (Ref. 19):

P- and V-operations are "indivisible actions"; i.e. if they occur "simultaneously" in parallel processes they are noninterfering in the sense that they can be regarded as being performed one after the other.

```
do i=1,N
      do j=1,M
            U(i,j)=U(i-1,j)+U(i,j)+U(i+1,j)+U(i,j-1)      (9)
      end do
end do
```

into the following parallel equivalent:

```
semaphore S(:,:)
...
doacross i=1,N
      do j=1,N
            if (i.ne.1) P(S(i,j))
            U(i,j)=U(i-1,j)+U(i,j)+U(i+1,j)+U(i,j-1)   (10)
            V(S(i+1,j))
      end do
end doacross
```

Doall loops can have synchronization instructions in their bodies as long as they do not require a particular allocation order or a minimum number of processors. This will be the case when the synchronization instructions are those used to create critical sections. For example, the loop:[9]

```
do i=1,N
      A(K(i)) = A(K(i)) + 1                              (11)
end do
```

is equivalent to the loop:

```
semaphore S(:)
...
doall i=1,N
      P(S(K(i))                                          (12)
      A(K(i)) = A(K(i)) + 1
      V(S(K(i))
end doall
```

Besides loop parallelism, microtasking has also been used for straight line parallelism when the execution time of each segment is relatively short.

In Table 1, a summary of the main features of several parallel Fortran dialects is presented. Fortran remains predominant as the supercomputer programming language. However, there is no lack of advocates for other languages. Foremost among the contenders are the functional languages; these include FP (Ref. 7), ID (Ref. 39), VAL (Ref. 1), SISAL (Ref. 36), and ParAlfl (Ref. 27). In functional programming there is no global state being modified by the program, but only functions mapping input values onto output values. Some have claimed that these languages are more appropriate for parallel processing due to the lack of side effects. However, as far as we know, there is today no high-quality implementation of any of these languages that can successfully compete with Fortran in the generation of efficient object code for supercomputers.

[9]Notice that in this loop, if several iterations operate on the same element of A, as happens when several elements of K(i) are equal, the order in which the iterations are done is not important since each iteration simply adds one to an element of A. However, it is important that no two iterations operating on the same element of A be done in parallel or the effect of some of these iterations will be lost. This sequentialization is taken care of by the next loop.

Language	Document Date	Vector Assignment Statements	Vectorizing Compiler	Multitasking	Microtasking
Illiac IV FORTRAN (Ref. 11)	1971	Control Vectors	NO	NO	NO
Glypnir (Ref. 33)	1972	Yes	NO	NO	
IVTRAN (Ref. 38)	1973	DO FOR ALL	YES	NO	NO
Vectran (Ref. 42)	1975	Triplets IDENTIFY WHERE	NO	NO	NO
BSP FORTRAN (Ref. 12)	1975	Triplets IDENTIFY WHERE	YES	NO	NO
HEP FORTRAN (Ref. 29)	1978	NO	NO	Hardware Supported	NO
FMP FORTRAN (Ref. 13)	1979	NO	NO	NO	Hardware Supported No Synchronization
FORTRAN 8X (Ref. 37)	1987	Triplets IDENTIFY WHERE	N/A	NO	NO
Fujitsu FORTRAN (Ref. 20)	1985	NO	YES	NO	NO
Convex FORTRAN	1985	NO	YES	NO	NO
IBM VS FORTRAN (Ref. 45)	1985	NO	YES	NO	NO
Cray CFT (Ref. 16)	1980	NO	YES	Cray Primitives	Software Supported Critical Regions
Alliant FORTRAN (Ref. 5)	1985	Triplets	YES	UNIX Primitives	Hardware Supported Cascade Synchronization (Implicit)
Sequent FORTRAN (Ref. 40)	1985	NO	-	UNIX Primitives	Software Supported Critical Regions Cascade Synchronization
EPEX/FORTRAN (Ref. 30)	1985	NO	-		Software Supported Barrier Synchronization

Table 1 Characteristics of Fortran Implementations

An area not widely explored so far is that of parallel symbolic computing. We believe that much more will be done in this area in the future. For this type of programming, parallel versions of Lisp (Refs. 21, 24 and 49), and Prolog (Refs. 15 and 47) are being developed. Also, some compiler techniques to parallelize Lisp programs have been developed (Refs. 25 and 26).

4 STATE-OF-THE-ART PROGRAMMING TOOLS

Restructuring and Interactive Restructurers

As we discussed above, many supercomputers include software for automatically extracting parallelism from what was originally sequential code. We will start this section by presenting several examples of parallelization. One of the simplest transformations is the one that can be performed when all iterations are independent of each other. The way to determine whether the loop iterations are independent is by computing a *data dependence graph*. We will not define this graph here; more information on dependence graphs and program parallelization may be found in Refs. (30) and (11). In this paper we will limit ourselves to a few examples of transformations.

All iterations in the following loop are independent:

```
do i=1,n
      if (A(i) .gt. 0) then
            B(i) = C(i) + D(i)                                    (13)
            E(i) = F(i)
      end if
end do
```

and therefore it can be transformed to:

```
where   (A(1:n) .gt. 0)
      B(1:n) = C(1:n) + D(1:n)
      E(1:n) = F(1:n)                                             (14)
end where
```

or to:

```
doall i=1,n
      if (A(i) .gt. 0) then
            B(i) = C(i) + D(i)
            E(i) = F(i)                                           (15)
      end if
end doall
```

or to:

```
doall i=1,n,K
      m = min(i+K-1,n)
      where (A(i:m) .gt. 0)
            B(i:m) = C(i:m) + D(i:m)
            E(i:m) = F(i:m)                                       (16)
      end where
end doall
```

Parallelizing loops is possible even when loop iterations are not independent — parallelizing compilers could transform do loops into doacross loops by inserting the appropriate synchronization instructions. For example, a parallelizing compiler could transform do loop (9) to doacross loop (10).

Sometimes secondary transformations are necessary before a loop can be parallelized. For example, the loop:

```
do i=1,n
      A = B(i) + C(i)
      D(i) = A + 1                                                (17)
end do
```

cannot be parallelized because all iterations use the variable A to store intermediate values.

A transformation called *scalar expansion* will make the iterations of the previous loop independent by changing A into an array:

```
do i=1,n
      AX(i) = B(i) + C(i)
      D(i) = AX(i) + 1                                            (18)
end do
A = AX(n)
```

Another important secondary transformation is *loop interchanging*. This transformation makes it possible to map either of the following two doubly-nested loops into the other:

```
do i=1,n
      do j=1,n
            A(i,j) = A(i,j-1) + 1                    (19)
      end do
end do

do j=1,n
      do i=1,n
            A(i,j) = A(i,j-1) + 1                    (20)
      end do
end do
```

If the input loop is the first one above, and the target machine is a vector machine, the compiler will first transform the first loop into the second and then vectorize the inner loop.

On the other hand, if the input loop is the second one and the target machine is a multiprocessor, the compiler will first transform the second loop into the first loop and then transform the outer loop into a `doall` loop. This is done to pay only once the overhead involved in starting the `doall` loop.

The final secondary transformation we will discuss is *blocking*. This transformation is used mainly for memory management. For example, assume a vector machine where all arithmetic machine instructions are register-to-register, and that the vector registers are 32 words long. The loop:

```
do i=1,n
      A(i) = B(i) + C(i)                             (21)
end do
```

can be transformed into:

```
do i=1,n,32
      do j=i,min(i+31,n)
            A(j) = B(j) + C(j)                       (22)
      end do
end do
```

The inner loop can be vectorized as follows:

```
do i=1,n,32
      m = min(i+31,n)
      A(i:m) = B(i:m) + C(i:m)                       (23)
end do
```

The vector operations can now be mapped into vector register instructions as shown next (`vr1`, `vr2` and `vr3` are 32-element vector registers)

```
do i=1,n,32
      m = min(i+31,n)
      vr1 = A(i:m)
      vr2 = C(i:m)
      vr3 = vr1 + vr2                                (24)
      A(i:m) = vr3
end do
```

Let us now discuss a more complex example. Assume a multiprocessor with a cache memory on each processor. Further assume that the cache (and thus the memory) is divided into blocks of K words each, and that data is only exchanged between memory and cache as whole blocks. Assume also that matrix columns are sequences of blocks (i.e.,

matrices are stored in column-major order and columns are much bigger than blocks).

Consider now the loop:

```
do i=1,N
      do j=1,N
            B(j,i) = A(i,j) + 1                    (25)
      end do
end do
```

A naive compiler might transform the outer loop into a doall without any other transformations, causing (1+1/K) block transfers between memory and caches for each assignment executed.

To improve this situation, the compiler could block both loops into groups of K iterations. This would have the effect of transposing the matrix by KxK submatrices or blocks, thus the name *loop blocking*. After blocking and interchanging loops, we end up with the loop:

```
do io=1,N,K
      do jo=1,N,K
            do i=io,io+K-1
                  do j=jo,jo+K-1
                        B(j,i) = A(i,j) + 1        (26)
                  end do
            end do
      end do
end do
```

If the outer loop is now transformed into a doall loop, the number of cache block transfers decreases to 2/K, a clear improvement over the naive approach when K is not small.

To conclude the work on this loop we need to block once more for vector registers, vectorize the innermost loops, and map into vector register instructions:[10]

```
doall io=1,N,K
      do jo=1,N,K
            do i=io,io+K-1
                  do j=jo,jo+K-1,32
                        m=min( j+31, n)
                        vr1 = A(i, j:m)
                        vr2 = vr1 + 1              (27)
                        B(j:m, i) = vr2
                  end do
            end do
      end do
end do
```

Even though for presentation purposes the previous examples were shown as source-to-source transformations, parallelization is most often performed inside the compiler, and usually the programmer is informed of the transformations only via annotated program listings. The annotations, when used in conjunction with information on execution time of program segments (known as a *program profile*), identify those segments of

[10] Some transformations like vectorization actually make the code easier to read. On the other hand, loop blocking is an example of a transformation that the user would prefer not to see — it certainly does not make the code any easier to read even though it makes it more efficient.

code for which the programmer, in his quest for speedup, should rewrite or expand with parallelization directives to the compiler. Two examples from Alliant Fortran are the commands `cvd$g cncall` and `cvd$g nosync`. The first one allows loops to be parallelized in the presence of subroutine calls.[11] The second command allows parallelization of loops even if several iterations assign values to the same memory location.

The existence of these annotations indicates that parallelization is different from traditional compiler optimizations. Thus, a compiler performs register allocation and usually common subexpression elimination, but it never informs the user on how successful it was in applying these transformations. The major difference between regular optimizing compilers and vectorizing/parallelizing compilers is that the benefits of vectorization are potentially higher, and either program rewriting or parallelization commands may be necessary to obtain efficient parallelism. An example of such a situation is provided by the transformation that takes a Fortran do loop and transforms it into a vector assignment statement. One piece of information required for this transformation is an analysis of the array subscripts inside of the do loop. This analysis can be performed when the subscripts are linear functions of the loop indices. When this analysis cannot be performed, vectorization is precluded. For example, the loop:

```
do i=1,100
     A(K(i))=A(K(i)) + 1.1                    (28)
end do
```

cannot be vectorized[12] if nothing is known about vector K. One way to get this information is via assertions. In this case, the compiler will vectorize only if the programmer asserts that $K(i) \neq K(j)$ whenever $i \neq j$.

Depending on the target machine, the annotations provided to the programmer may vary in complexity. In some cases it may be appropriate to show the programmer a source code version of the parallelized program. This approach is followed by those parallelizers that perform source-to-source translation such as Parafrase (Ref. 31), KAP (Refs. 17 and 18), VAST (Ref. 9), and PFC (Ref. 3). The output of these restructurers includes some form of parallel constructs. Since no standards exist for such constructs, there is no uniformity even though some form of vector extensions from Fortran 8x are frequently adopted.

The input parallelization commands can also be replaced by parallel constructs. Thus, instead of specifying that a loop may be vectorized, the programmer may write a vector assignment statement. However, portability suffers when parallel extensions are used since there is no standard for those extensions.

In the past, vectorizing compilers have been considered only in the context of dusty decks[13]. We believe that restructurers (source-to-source translators) are also useful in

---

[11] When compilers are faced with subroutine calls inside of loops, they usually assume the worst — that the loop cannot be vectorized or multiprocessed. Recent work (Refs. 10, 14, 28, and 52) suggests that some interprocedural analysis can be done to permit transformations in some cases. But user assertions are still going to be important.

[12] By vectorizing we mean transforming loop (28) into the single statement

```
A(K(1:100)) = A(K(1:100)) + 1.1
```

We should point out that even if nothing is known about K we could transform (28) into a sequence of vector statements, where the first ones do some sort of runtime dependence testing.

[13] The term *dusty deck* refers to old (dusty) programs that need to be compiled without human rewriting, either because the cost of manpower needed for the rewriting is prohibitive, or because nobody understands the programs any more. Some people feel that the only use for sophisticated restructuring compilers is for processing dusty decks — that new languages allowing the explicit expression of parallelism will obviate the need for these compilers. One look at the results of loop blocking above should convince anyone that even if programmers can use explicit parallelism, there are some optimizations better left to the compilers.

other contexts. Specifically, a restructurer could be used to free the programmer (at least to some extent) from performance considerations and let him concentrate on correctness. We could, therefore, conceive of programming parallel computers as a two-step process. First, a program would be written and tested. Once the programmer was convinced of its correctness, the program would be transformed into an efficient version through automatic means. Clearly, things will not always work out in this way since achieving efficiency might involve changing the program in ways beyond the capability of current restructurers. However, we believe that this two-step process could often be applied.

In the process of restructuring, interaction with the user may be needed due to one of the following reasons:

- A certain transformation is valid only if the user supplies an assertion. The vectorization of loop (28) is an example of this situation.
- The restructurer needs information from the user to decide how to transform a construct. For example, a vector assignment where only some of the array elements are to be assigned may be transformed into a sequence of vector operations including gather/scatter operations or into vector operations that mask some of the assignments. Knowing the density of the array elements to be assigned is necessary in this case (See Ref. 20).
- The restructurer may not have a model of the target machine, and therefore it will be unable to decide by itself what transformations to apply even if the program behavior is known at compile time.

Most of the restructurers today are batch restructurers. Some of them interact with the programmer (also in batch mode) by requesting information in the listing and accepting comment cards with information supplied by the programmer. However, interactive restructurers have clear advantages, and are beginning to emerge.

Debuggers

Much has been done in the area of symbolic debuggers over the last ten years. Many user friendly tools such as Berkeley Unix's **dbx** (Ref. 53) and Apollo's **debug** (Ref. 6) have done an excellent job of providing an environment that allows controlled probing and analysis of application programs. Debuggers of this type typically support a set of tools that can be used to determine the state of an object program at any point in its execution. These tools include the facilities for setting breakpoints, monitoring, examining, and tracing variables (which can be symbolically referenced), and single-stepping the target program. This set of functions is usually sufficient to allow the user to determine the unique state of a single processor application within the context of the original source program. Supercomputer applications, however, present a more difficult problem to the debugging programmer. These programs use vector and/or multitasking parallelism in order to achieve their high computation speed. Parallelism in programs introduces new wrinkles that make it difficult to extend serial debugging tools to the parallel domain. Additionally, as mentioned earlier, the optimizing compiler tools that usually accompany the supercomputer hardware do extensive restructuring of the original source program, widening the gap between the user's perception of the application and the run-time representation.

Vector parallelism by itself is not inherently complicated. Languages that include vector constructs exist today, and compilers for these languages can map high level vector constructs directly into vector instructions for the pipelined vector architectures. Programs that explicitly use only vector parallelism can be analyzed with traditional debuggers having minimal extensions, since these programs still execute through a single stream of statements at one time. The problem arises when serial programs are passed through vectorizing restructurers. These optimizers can significantly change the

appearance of the original source code, making it difficult for a user to reconcile the edit-time and run-time states of his program. For this reason, the reporting of run-time errors may not make sense when returned to the user, forcing him to recompile his program without optimization and rerun the application in order to help isolate problems.

Multitasking applications present a more serious challenge to debugger technology (as well as ideology). In the parallel execution domain, the concept of a breakpoint is not clear. In the parallel environment the notions of global and local effects must be considered. A local effect is one that is administered to a small group (possibly one) of processors, whereas a global effect is one that is administered to all processors related to the execution of a particular program. A breakpoint applied to one portion of code executing on one processor may or may not be expected to halt all other processors executing the same code.[14] Furthermore, should a global effect for that breakpoint be desirable, the question of granularity must be resolved, namely, how quickly after the breakpoint is reached can all of the other processors be stopped? A similar problem exists with respect to the variable name space. Tracing and examining specific variable names becomes more tedious when several processors are executing, each with its own copy of the same routine (and hence the same list of local variables). Tracing of such variables must be qualified with additional information that identifies the processor or group of processors of interest.

The most challenging aspect of parallel debugging is the timing conflicts introduced by interacting, independently running processors. The series of states through which a serial program passes is not time dependent and is therefore repeatable, providing the opportunity for an unlimited number of reruns in order to localize run-time anomalies. The set of states through which a parallel program passes is dynamic and very sensitive to the speed at which each processor is progressing. For this reason, program errors may surface infrequently. Furthermore, these timing or synchronization errors might be completely masked when software debugging instrumentation is inserted into the code (thus changing the run-time image).

Another debugging problem is the nature of supercomputer application programs. These programs tend to manipulate large quantities of single- and double-precision floating-point numbers to perform their tasks. Finding errors in output listings from lengthy computations can be user-intensive and time-consuming. New methods of rendering this information in the form of graphic images must be used to present large volumes of information in a concise manner.

Many solutions to the parallel debugging problem are starting to appear in industry and academia. One such solution is **Pdbx** developed by Sequent Computer Systems, Incorporated (Ref. 46). **Pdbx** is an enhanced version of **dbx** that supports debugging of multiple process applications on Sequent's shared memory multiprocessor machine. In addition to the functionality of **dbx**, **Pdbx** supports the debugging of multiple Unix processes. Supported are such features as breakpoints for one or more processes, independent examination and tracing of individual processes, and the use of multiple terminals or "windows" for monitoring multiple processes. While **Pdbx** provides no facilities to control the repeatability of a parallel program, it does extend the functions of a traditional serial symbolic debugger to provide some tools for probing the execution of parallel programs.

Instant Replay,[tm] developed at the University of Rochester, is another debugging environment targeted at helping users debug parallel programs on the BBN Butterfly (Ref. 34). **Instant Replay** attacks the repeatability problem by regulating and recording access to shared data objects. By introducing some small run-time overhead (variable,

[14] CSRD is currently researching the question of whether global effects are necessary or desirable when debugging parallel programs.

but as low as one to ten percent in some applications), **Instant Replay** attaches aging information to all shared objects and records revision numbers as these objects are updated and disseminated. In addition to recording this revision information, the run-time system has the ability to "replay" the application while insuring the same access sequences to shared objects. This gives the programmer the capability to perform the cyclic rerunning necessary to do incremental debugging on a parallel machine.

The future will probably bring developments in several areas in response to the increased challenge of constructing and debugging parallel programs. Hardware enhancements represent one necessity. New supercomputer designs will continue to incorporate an increasing amount of instrumentation to support run-time monitoring and control. This special-purpose hardware is necessary to monitor the execution history of parallel programs in a non-intrusive way. Additionally, low-level hardware support can be used to effect a parallel breakpoint mechanism that cannot be cleanly achieved in software.

Other tools to be expected in the future include pre-compilation and post-run analysis tools. Designers of parallel software will be able to use these tools to analyze interprocessor communication and cite coding sequences that can be potentially time dependent. Such an analysis system is being developed at the University of Illinois at Urbana-Champaign (Ref. 4). The system consists of a diagnostic compiler that can warn the user of source code sequences that contain guaranteed race conditions as well as potential race conditions that cannot be determined with certainty at compile time. To help ascertain the behavior of indeterminate race conditions, the compiler will automatically insert the appropriate instrumentation into the object code to investigate data reference behavior. Trace data generated by this instrumentation can then be inspected (both manually and automatically) to help determine the status of potential timing hazards recognized before execution. Systems such as this serve not only to assist users in locating nondeterministic code sequences in parallel applications, but also to help users improve program performance. By allowing the compiler to instrument an application in areas of uncertain race conditions, the user can potentially learn of assertions he could add to his code that would eliminate conservative assumptions that might normally be made by the restructuring compiler.

Improvements in development environments will also help to ease the burden of using supercomputers effectively. Knowledge-based systems could play a major role in the implementation of an error-free program. By guiding the user through the selection from a library of optimized numerical kernels, expert systems can aid the user in locating reusable, bug-free code. This is one way the environment can help to reduce the potential number of software errors during development.

Another way is through the use of new knowledge-based debugging tools. An intelligent parallel debugging system could work in conjunction with the system compilers and run-time monitoring system to ascertain the required information for a processor-time graph (Figure 1). Once an error is recognized in the program, the programmer could invoke a debugging expert that would ask a series of questions relating to the error to help the user isolate the problem. In the case of an intermittent synchronization error, the expert system could reference the processor-time graph and supply a mapping of execution time to source code lines to aid the user in looking for programming errors. In the case of Figure 1, the programmer would be directed to review the code that executes between times t_3 to t_4 and times t_5 to t_6 since maximum parallelism occurs during these times and is therefore likely to be the cause of intermittent synchronization problems.

Finally, continuing improvements in restructuring compiler technology will reduce the pressure on individuals to find and implement parallelism in application programs. While parallel programming must be encouraged in order to build applications that achieve the highest levels of performance, automatic optimization systems will allow more casual users to develop and debug an application in a serial, reproducible environment — leaving the

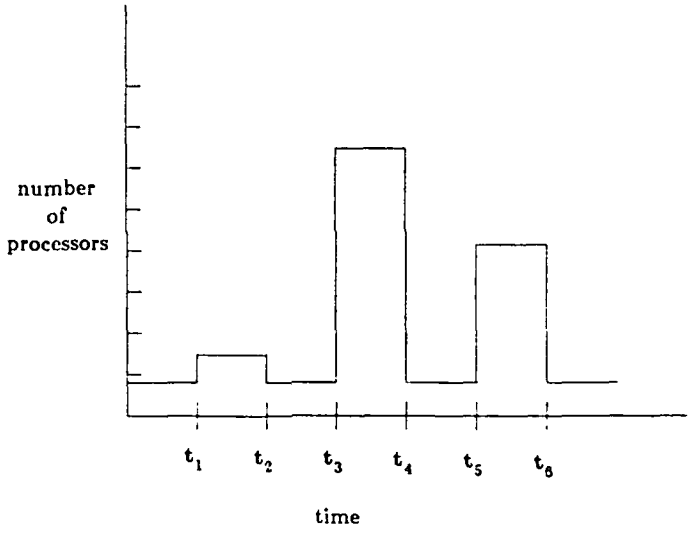


Figure 1 Sample processor-time graph

correct parallelization transformations to the compiler.

Performance Evaluation

Performance evaluation on any machine has traditionally been a difficult and much neglected task. Supercomputers worsen the problem with a complicated arsenal of hardware and software additions that require new levels of understanding by the user. Many of the architectural features used by supercomputer designers to gain computational speed are the same features that make program performance difficult to track. Pipelined vector arithmetic units, shared caches, interconnection networks, and shared memories are just a few aspects of supercomputer systems that can be responsible for a wide variance of execution times for a particular application, depending on their operational efficiency. These architectural components are complex and interrelated, resulting in a run-time environment that is difficult to trace and analyze.

State-of-the-art performance evaluation tools for existing single-processor machines are helpful when extended into the parallel domain, but remain too simplistic to do a complete job. These utilities help programmers understand the run-time resources consumed by an application at a high level, without providing insight into the machine level interaction that might be involved. The Unix operating system supports performance analysis for programs written in Fortran, C, and Pascal through a set of profiling tools and system timer calls. One of the profiling tools is the **gprof** (Refs. 53, 22, 23) utility which allows users to accumulate call counts and execution times on a subroutine basis by sampling the program counter of the running application at regular intervals. This information is collected through the use of special run-time code inserted into the object module on demand by the user at compile time. The profiling technique is very useful for isolating specific routines that account for major portions of an application's execution time, but does little for helping the user to improve efficiency and utilization once the trouble areas have been located.

For example, the following code fragment:

```
do 10 I = 1, 100
        do 10 J = 1, 100
                do 10 K = 1, 100
                        A(I, J, K) = 0                    (29)
                end do
        end do
end do
```

is an example of a loop that will generate an excessive number of page faults when executed in a virtual memory system. The reason for the performance problem is that the array is referenced in the "wrong" order. With Fortran arrays, element A(1,1,1) is adjacent in memory to element A(2,1,1), not A(1,1,2). For that reason, each write to A(I,J,K) may require a disk access instead of a simple memory write. In this example, knowing that the routine which contains this loop is responsible for a significant percentage of the execution time for a particular application is useful but not sufficient. Inexperienced programmers who have not seen this effect may be unable to correct performance deficiencies of this sort. The analysis given by the run-time system should include more detailed information about the nature of the delays incurred in the designated routine along with suggestions about whether the observed performance is "reasonable."

While the above example represents a simple problem that can also be seen in uniprocessors, other examples for parallel machines can be more subtle. Consider, for example, an application that applies a set of n processors using a shared cache to do a computation on n individual, independent arrays. Depending on the size of the arrays, the access pattern of each processor, and the cache algorithm used by the memory system, the speedup seen for this application could be anywhere from n down to numbers less than 1.[15] Speedups of less than 1 can result from each of the n processors overwriting cache blocks that have just been loaded by one of the other processors. If perfectly timed, each processor's memory reference would result in a cache miss, thus nullifying the usefulness of the cache.[16] While the situation described is a pathological case, some interference effects do exist in the caching system that can impact performance. Additionally, other aspects of the machine operation such as contention in the interconnection network also impact the program's performance. Complete analysis of the performance of applications in the presence of these effects requires the ability to capture this information.

As was the case with debugging, restructuring compilers complicate the effort of tuning supercomputer programs. Again, the user is faced with the problem of reconciling run-time trace and performance information with source code listings that do not completely match. More recent optimization techniques such as subroutine expansion (Refs. 28, 35) pose some of the more difficult problems, since many performance tools tend to collect data at the subroutine level and this transformation erases that modularity. Subroutine expansion enlarges the granularity of the monitored program, giving the user less detailed information about the nature of the run-time performance.

Another aspect of computing in general (and supercomputing in particular) that is troublesome for performance evaluation is multiprogramming. Since investments in supercomputer hardware can be quite large, there is a strong incentive to achieve maximum utilization of available machine time. Utilization of supercomputers is usually enhanced through multiprogramming. This causes problems for the performance analysis

---

[15] Speedup is T_1 / T_n, where T_1 is the time required to execute the application serially and T_n is the time required to execute the application using n processors. Speedups of less than 1 indicate that the application runs slower in the parallel environment than it would have serially.

[16] In fact, since the caching mechanism introduces some overhead, the resulting application might run slower than it would with a single processor system with no cache.

system by complicating the hardware and software instrumentation. Multiprogrammed systems require enhanced performance analysis instrumentation in order to do the necessary accounting for multiple jobs. For job swaps and operating system calls, more context information must be saved to insure integrity of individual job accounting. Hardware and software probes must be turned off and on during context switches instead of running continuously as in a monoprogrammed environment.

Multiprogramming also causes problems for the user attempting to improve an application's execution performance. Applications running in monoprogrammed environments are easily evaluated by examining the apparent execution time of execution or "wall clock" time. Users optimizing applications in such an environment need only make changes and execute again, comparing the new wall clock time to that of previous runs. While wall clock time is a useful metric to gauge the performance of applications in these monoprogrammed environments, it is not useful in understanding the performance of an application in a multiprogrammed system. System loading, scheduling algorithms, resource availability, and other factors uncontrollable by the user all contribute to the apparent execution time of an application. The user will find that two successive runs of an identical program could vary greatly.

Future programming environments will include many enhancements to the hardware/software configurations that are offered by supercomputer vendors if application programs are to perform efficiently on their machines. First, hardware enhancements will be necessary to achieve many of the performance evaluation goals currently envisioned. Some manufacturers have already realized the need for including such specialized hardware as part of their standard machine configurations. Cray Research, Inc. manufactures a machine called the X-MP which includes a hardware performance monitor that comprises a set of counters that can monitor certain hardware-related events (Ref. 32). These counters track events such as floating-point operations, instruction fetches, I/O and CPU memory references, and vector operations on a per-CPU basis.[17] As supercomputing experience accumulates, the heightened awareness of the need for performance information will drive other manufacturers to provide a basic set of hardware instrumentation that can be used for performance and correctness tracking.

Perhaps the greatest potential for improvement in the area of performance evaluation is that of presentation techniques. While data capturing facilities in system hardware and software are evolving, innovations in the area of data rendering are slow in coming. The volume of run-time trace data that could be of interest in a parallel execution environment is too massive to represent in raw form. Of particular interest are areas of interaction between multiple processors that are synchronizing in some way. A detailed analysis of this environment could require an extensive log of time-stamped accesses. Such a log is usually not practical to review, possibly consisting of several hundreds of pages of entries. More interesting techniques involving concise graphic representations must be developed to make this information more usable.

New innovations in the area of pre-compilation and pre-execution performance analysis tools can also be expected. These tools might take several forms to aid the user at different times during the program development cycle. One such tool might be a program analyzer that could evaluate the use of system library routines and present estimates of the run-time performance of an application based on past execution statistics of the library kernels and the extent of their usage. Other tools might look at generated assembly code and make predictions about execution speed based on the density of vector computation opcodes versus that of scalar, control, and other "glue" opcodes.

---

[17] The Cray X-MP is available in multiple CPU configurations.

As with debugging, the problem of performance evaluation can lessen as compiler technology continues to improve. The problem of tuning application programs should gradually become a higher level concern than it is today. Compilers will continue to find new ways of exploiting parallelism at low levels, while applications programmers are freed to concentrate on higher level algorithm design issues. Knowledge gained by the performance analysts today will be incorporated in tomorrow's compilers. As the effects of caching and memory interconnection networks are better understood, heuristics for better transformations can be built into optimizing compilers. These compilers should also be adaptive — able to generate efficient code for many different vendors' machines in a particular class and dependent only on a list of important parameters such as cache size, vector register lengths, number of processors and others. Additionally, these compilers might be able to use data collected by performance monitoring systems in order to further refine compile-time optimizations. The current static decisions about optimizations might be dynamic in the future — based on information about the eventual run-time environment (such as system loading). In this environment, the compiler system will migrate toward an expert system model, soliciting information from the user and statistics databases in order to provide optimal execution for a wide range of applications.

Integration

With the number and complexity of software development tools increasing, the need for an integrated environment is becoming increasingly necessary in the high speed computing arena. A graphics-based scientific programming environment with an integrated software productivity tool kit has many things to offer to the supercomputer programmer that cannot easily be offered by the conventional software development tools being used today.

First, the programming environment should provide a consistent user interface paradigm across the entire range of supported tools. Casual or infrequent users are not likely to spend extensive sessions with user manuals learning the idiosyncrasies of several tools. Rather, users will become frustrated with the system's complexity and will resort to functioning at the easiest possible level, thereby minimizing his effort (and possibly his efficiency and productivity).

A well-structured environment should consist of a single interface style through which all packages are accessed. The screen images seen by the user during program editing should be the same as the images seen during debugging and program optimization. Once the user is fluent with one aspect of the system, several functions of the system should be usable with minimal additional effort. The efficacy of such an approach can be seen by the ease of use and popularity of extant integrated window-based systems such as the Apple MacIntosh.

Additionally, the user interface should support a graphical as well as a textual representation for programs. Indeed, source text will always be viewable and editable by the programmer, but higher level abstractions such as static subroutine call graphs and task/process graphs are useful in understanding overall program structure more readily. Just as graphic images can be used to render a concise representation of large volumes of output data, graphic program structures can be a useful tool for the programmers wishing to elide much of the source-level implementation details in favor of perusing a terse representation of an application's architecture.

The Center for Supercomputing Research and Development at the University of Illinois is currently developing an environment that supports such a programming model. The environment, named Faust, is targeted at integrating several software development tools through a common window-based interface. For example, a user wanting to develop an application at the source-code level may bring up a textual window and enter Fortran source using a conventional text editor (Figure 2). If the user would rather see the

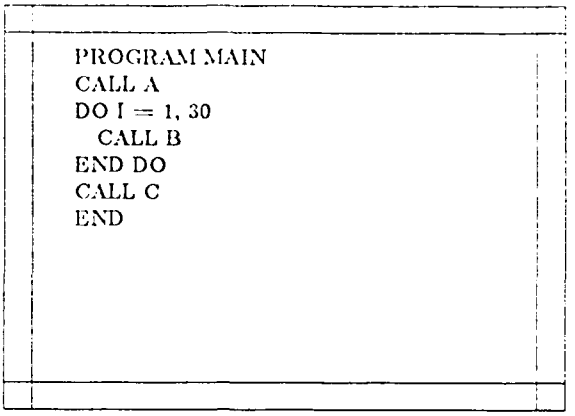


Figure 2 Simple application being examined at the source code level

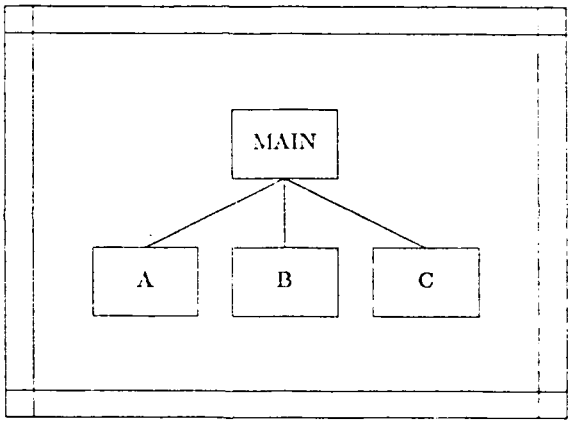


Figure 3 Same application at the subroutine interconnect level

application at a higher level, an "unzoom" function can be invoked to bring up the corresponding subroutine interconnection graph (Figure 3). Faust can automatically create the subroutine call graph (if it does not already exist) from source code; however, if desired, the user may do the original editing at the graphic level of abstraction and associate source code for each "block" as the implementation proceeds. Faust also supports other levels of detail including process graphs that represent parallelism as well as data-dependence graphs for aiding interactive restructuring.

The concept of detail hiding is the same rationale used to justify high-level languages. Programmers usually want to concentrate on a solution to an application problem in more abstract terms than is possible through the use of assembly language. For this reason, high-level languages create the "language environment" in which the programmer works. Hiding the details, however, is not always desirable. In some operations, the programmer may require the detailed information to correct deficiencies in his application (especially with respect to debugging and performance improvement). To support this facility,

several Unix-based compilers often include an option that allow a user to see the assembly code generated from his source files. This gives the user the ability to work at the higher level abstraction during normal use as well as to analyze machine level details when necessary. The superenvironment builds on this idea, providing more levels of abstraction that are completely controllable by the user. While this issue of multiple levels of detail is not specific to the programming environments of supercomputers, the additional complexity of multiple streams of execution makes the abstraction even more desirable.

The scientific programmer's environment needs to be flexible in a number of different ways. First, the environment must support users of varying levels of expertise. Engineering-oriented users are likely to want to concentrate on solving applications problems — not performance problems. The environment should support this group of users with an array of automatic restructuring tools and electronic experts that can shoulder most of the burden of achieving execution efficiency while the user focuses on his algorithms and application. Numerical analysts and systems programmers, however, will expect the environment to provide more fundamental tools for scrutinizing the more subtle aspects of machine operation in order to retrieve the low level data they need to fine tune system libraries. Somehow the environment must support both ends of the spectrum in a unified manner.

Second, the programming environment should be able to support multiple types of machines. Many styles of new machines will be developed over time and the life of an application program will succeed several machine architectures. Additionally, many users support applications on multiple machines at the same time, frequently moving applications between them as required. For these reasons, the environment should be adaptable enough to support a production application on several vendors' architectures without requiring significant effort from the user. This can be achieved by designing an extensible interface through which remote utilities (such as vendor-specific optimizing compilers) can be attached while maintaining the same dialogue and appearance to the user. The local utilities should also be designed to work according to heuristics developed for general architectural characteristics rather than machine specific idiosyncrasies (although, as mentioned above, numerical analysts will want to take advantage of machine-specific phenomena when building heavily used kernels). For example, a restructuring compiler that does transformations for a generic vector processor with vector register length n can be useful for a number of different machines just by supplying the appropriate n for the machine of interest.

Finally, the environment should support a wide variety of language domains. While Fortran is certainly necessary, languages such as C, Ada, Pascal, Val, and others must also be considered. Future developments are also likely to include languages of a more symbolic nature. Interactive environments built on systems such as Maxima (Ref. 44) and Reduce (Ref. 51) could provide a very useful function, offering a higher level of communications to scientific users who would prefer to express problems in a representation that more closely resembles mathematical notation than procedural source code. Ultimately, programming environments will evolve to transcend the mundane details of traditional programming, allowing scientists and engineers to converse in a language more familiar to them while the environment fills the gap between the symbolic representations and the encoding required by the underlying hardware.

The scientific programming environment is well suited to the workstation hardware offered by manufacturers such as Sun, DEC, Apollo and others. These nodes consist of bitmapped graphics screens attached to a 32-bit microprocessor running Unix. While running the environment's "front end" software on another host (the workstation) poses some technical problems with respect to implementation of control and communication links, this configuration offers the ability to run screen intensive user interaction support functions locally which provides several benefits. First, the user need not use expensive

supercomputer hardware to service functions such as text editing and graphics management. Second, keeping much of the functionality within the workstation helps promote the desired goal of supercomputer vendor independence. This common front end also promotes familiarity across systems through the enforcement of a common user interface between the application programmer and the supercomputer. Finally, having local intelligence in the workstation gives the user more consistent response from day to day.

5 ACKNOWLEDGEMENT

This work was supported in part by the National Science Foundation under Grant Nos. US NSF DCR84-06916 and US NSF DCR84-10110, the US Department of Energy under Grant No. US DOE DE-FG02-85ER25001, the United States Air Force under Grant AFOSR-85-0211, and by a donation from the IBM Corporation.

6 REFERENCES

1. Ackerman, W. B., and Dennis, J. B., "VAL: A Value-Oriented Algorithmic Language," Report No. TR-218, June 1979, Laboratory for Computer Science, M.I.T., Cambridge, MA.

2. Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," Computer, Vol. 19, No. 8, Aug. 1986, pp. 26-34.

3. Allen, J. R., and Kennedy, K., "PFC: A Program to Convert Fortran to Parallel Form," Report No. MASC-TR82-6, Mar.1982, Rice University, Houston, TX.

4. Allen, T., and Padua, D., "Debugging Fortran on a Shared Memory Machine," to appear in Proceedings of the 1987 IEEE International Conference on Parallel Processing, 1987.

5. Alliant Computer Systems Corporation, "FX/FORTRAN Language Manual," No. 302-00002-B, Jan. 1986, Alliant Computer Systems Corporation, Littleton, MA.

6. Apollo Computer, Incorporated, "DOMAIN Language Level Debugger Reference," No. 001525, 1985, Apollo Computer, Incorporated, Chelmsford, MA.

7. Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, Vol. 21, No. 8, Aug. 1978, pp. 613-641.

8. Booth, M., and Misegades, K., "Microtasking: A New Way to Harness Multiprocessors," Cray Channels, Vol. 8, No. 2, Summer 1986, pp. 24-27.

9. Brode, B., "Precompilation of Fortran Programs to Facilitate Array Processing," Computer, Vol. 14, No. 9, Sept. 1981, pp. 46-51.

10. Burke, M., and Cytron, R., "Interprocedural Dependence Analysis and Parallelization," Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN No. 21, Vol. 7, July 1986, pp. 162-175.

11. Burroughs Corporation, "Illiac Fortran Specification," No. 66106, Dec. 1970, Burroughs Corporation, Paoli, PA.

12. Burroughs Corporation, "Burroughs Scientific Processor (BSP) Vector Fortran Preliminary Specification," 1975, Burroughs Corporation, Paoli, PA.

13. Burroughs Corporation, "Numerical Aerodynamic Simulation Facility Feasibility Study," Mar. 1979, Burroughs Corporation, Paoli, PA.

14. Callahan, D., Cooper, K. D., Kennedy, K., and Torczon, L., "Interprocedural

Constant Propagation," Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN No. 21, Vol. 7, July 1986, pp. 152-161.

15. Clark, K. L., and Gregory, S., "Parlog: Parallel Programming in Logic," ACM Transactions on Programming Languages and Systems, Vol. 8, No. 1, Jan. 1986, pp. 1-49.

16. Cray Research, Inc., "Cray X-MP Multitasking Programmer's Reference Manual," Publication No. SN-0222, Oct. 1986, Cray Research, Inc., Mendota Heights, MN.

17. Davies, J., Huson, C., Macke, T., Leasure, B., and Wolfe, M., "The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer," Proceedings of the 1986 IEEE International Conference on Parallel Processing, 1986, pp. 833-835.

18. Davies, J., Huson, C., Macke, T., Leasure, B., and Wolfe, M., "The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer," Proceedings of the 1986 IEEE International Conference on Parallel Processing, 1986, pp. 827-832.

19. Dijkstra, E. W., "The Structure of the T.H.E. Multiprogramming System," Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 341-346.

20. Fujitsu Limited, "Amdahl VP/Application Development System Fortran 77/VP User's Guide," Publication No. MC-142006, July 1986, Amdahl Corporation, Sunnyvale, CA.

21. Gabriel, R. P., and McCarthy, J., "Queue-Based Multiprocessor Lisp," Proceedings of the 1984 ACM Conference on Lisp and Functional Programming, Aug. 1984, pp. 25-44.

22. Graham, S. L., Kessler, P. B., and McKusik, M. K., "Gprof: a Call Graph Execution Profiler," Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, Vol. 17, No. 6, June, 1982, pp. 120-126.

23. Graham, S. L., Kessler, P. B., and McKusik, M. K., "An Execution Profiler for Modular Programs," Software - Practice and Experience, Vol. 13, 1983, pp. 671-685.

24. Halstead, R. H., "Multilisp: A Language for Concurrent Symbolic Computation," ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, Oct. 1985, pp. 501-538.

25. Harrison, W. L., "Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor," Report No. 565, Mar. 1986, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL.

26. Harrison, W. L., and Padua, D. A., "Representing S-Expressions for the Efficient Evaluation of Lisp on Parallel Processors," Proceedings of the 1986 IEEE International Conference on Parallel Processing, 1986, pp. 703-710.

27. Hudak, P., "Para-Functional Programming," Computer, Vol. 19, No. 8, Aug. 1986, pp. 60-70.

28. Huson, C. A., "An In-line Subroutine Expander for Parafrase," Report No. 82-1118, December, 1982, M.S. thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.

29. Kowalik, J. S., Parallel MIMD Computation: HEP Supercomputer and Its Applications, The MIT Press, Cambridge, MA, 1985.

30. Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. J., "Dependence Graphs and Compiler Optimizations," Proceedings of the 8th ACM Symposium on

Principles of Programming Languages (POPL), 1981, pp. 207-218.

31. Kuck, D. J., Kuhn, R. H., Leasure, B., and Wolfe, M. J., "The Structure of an Advanced Retargetable Vectorizer," Tutorial on Supercomputers: Design and Applications, IEEE Press, New York, NY, 1984, pp. 168-178.

32. Larson, J., "Cray X-MP Hardware Performance Monitor," Cray Channels, Winter, 1986, pp. 18-19.

33. Lawrie, D. H., Layman, T., Baer, D., and Randal, J. M., "Glypnir- A Programming Language for Illiac IV," Communications of the ACM, Vol. 18, No. 3, Mar. 1975, pp. 157-164.

34. LeBlanc, T. J., and Mellor-Crummey, J. M., "Debugging Parallel Programs with Instant Replay," IEEE Transactions on Computers, Vol. C-36, No. 4, April, 1987, pp. 471-482.

35. Loveman, D. B., "Program Improvement by Source-to-Source Transformation," Journal of the ACM, Vol. 24, No. 1, Jan. 1977, pp. 121-145.

36. McGraw, J., Skedzielewski, S., Allan, S., Grit, D., Oldehoeft, R., Glauert, J., Dobes, I., and Hohensee, P., "SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual," Report No. M-146, Revision 1, Mar. 1985, Lawrence Livermore National Laboratory, Livermore, CA.

37. Metcalf, M., "Fortran 8X — The Emerging Standard," ACM Fortran Forum, Vol. 6, No. 1, April 1987, pp. 28-47.

38. Millstein, R. E., "Control Structures in Illiac IV Fortran," Communications of the ACM, Vol. 16, No. 10, Oct. 1973, pp. 621-627.

39. Nikhil, R. S., Pingali, K., and Arvind, "Id Noveau," Computation Structures Group Memo No. 265, July 1986, Laboratory for Computer Science, M.I.T., Cambridge, MA.

40. Osterhaug, A., Guide to Parallel Programming on Sequent Computer Systems, Sequent Computer Systems, Incorporated, Beaverton, OR, 1985.

41. Padua, D. A., and Wolfe, M. J., "Advance Compiler Optimizations for Supercomputers," Communications of the ACM, Vol. 29, No. 12, Dec. 1986, pp. 1184-1201.

42. Paul, G., "VECTRAN and the Proposed Vector/Array Extensions to ANSI FORTRAN for Scientific and Engineering Computation," Report No. RC 9223 (#40515), Jan. 1982, IBM T.J. Watson Research Center, Yorktown Heights, NY.

43. Paul, G., and Wilson, M. W., "The Vectran Language: An Experimental Language for Vector/Matrix Array Processing," Report No. G320-3334, Aug. 1975, IBM Palo Alto Scientific Center, Palo Alto, CA.

44. Pavelle, R., "MACSYMA: Capabilities and Applications to Problems in Engineering and the Sciences," Applications of Computer Algebra, Kluwer Academic Publishers, Norwell, MA, 1985, pp. 1-61.

45. Scarborough, R. G., and Kolsky, H. G., "A Vectorizing Fortran Compiler," IBM Journal of Research and Development, Vol. 30, No. 2, Mar. 1986, pp. 163-171.

46. Sequent Computer Systems, Incorporated, "Dynix PDBX Debugger User's Manual," No. 1003-42756, May, 1986, Sequent Computer Systems, Incorporated, Beaverton, OR.

47. Shapiro, E., "Concurrent Prolog: A Progress Report," Computer, Vol. 19, No. 8,

Aug. 1986, pp. 44-58.

48. Smith, B. J., "A Pipelined, Shared Resource MIMD Computer," Proceedings of the 1978 IEEE International Conference on Parallel Processing, 1978, pp. 6-8.

49. Steele, G. L., and Hillis, W. D., "Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing," Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, Cambridge, MA, Aug. 1986, pp. 279-297.

50. Stone, J. M., Darema-Rogers, F., Norton, V. A., and Pfister, G. F., "Introduction to the VM/EPEX FORTRAN Preprocessor," Report No. RC 11407 (#51329), Sept. 1985, IBM T.J. Watson Research Center, Yorktown Heights, NY.

51. The Rand Corporation, "Reduce User's Manual," No. CP78, Santa Monica, CA, 1985.

52. Triolet, R., Irigoin, F., and Feautrier, P., "Direct Parallelization of Call Statements," Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN No. 21, 7, July 1986, pp. 176-185.

53. University of California, UNIX User's Manual, Reference Guide--4.2 Berkeley Software Distribution, Computer Science Division, University of California, Berkeley, CA, 1984.

END

DATE

FILMED

5-88

DTIC